

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б10-мм

Тестирование аппроксимаций стандартной библиотеки Java в USVM

Филимонов Игорь Олегович

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Д. А. Мордвинов

Консультант:
инженер-исследователь кафедры системного программирования, М. П. Костицын

Санкт-Петербург
2024

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Символьное исполнение	6
2.2. USVM и аппроксимации	7
3. Описание решения	8
3.1. Реализация тестовой системы	8
3.2. Тесты	9
4. Тестирование	10
Заключение	11
Список литературы	12

Введение

Тестирование является важным элементом разработки программного обеспечения. Отчасти именно оно превращает обычные программы в программные продукты, которые могут применяться в индустрии и приносить пользу заказчику. Однако, с увеличением сложности программного обеспечения его тестирование также становится сложнее, так как в идеале оно должно покрывать всевозможные варианты использования ПО, даже те, вероятность появления которых в реальности крайне мала. Это послужило причиной развития различных техник автоматической генерации тестов, одной из которых является символьное исполнение [5].

Основой символьного исполнения является запуск программ на абстрактных входных данных, называемых символьными, что позволяет исследовать всевозможные пути исполнения (например, в случае конкретных данных при достижении оператора `if-else` исполнение продолжится только в одной ветке в зависимости от выполнимости условия в `if`, в то время как символьное исполнение будет исследовать оба варианта). Простое в теории, символьное исполнение едва ли эффективно реализуемо в чистом виде на практике. Одними из причин являются так называемый взрыв количества путей, а также проблемы со взаимодействием с внешним окружением программы (как вариант, со сторонними библиотеками).

Факт губительности этих причин для символьного исполнения осложняется их распространенностью. Так, например, в стандартной библиотеке Java, которая является неотъемлемой частью работы почти любого Java программиста, эти 2 причины проявляются особенно ярко в силу ее крайней оптимизированности, что повышает ее эффективность при обычном исполнении, но возрастающая при этом сложность кода делает ее едва ли пригодной для символьного исполнения за приемлемое количество времени.

Одним из вариантов смягчения влияния этих факторов на символьное исполнение является использование аппроксимаций. Аппроксима-

ция — это упрощенный вариант кода программы, который подменяет оригинал во время исполнения. Упрощение может достигаться различными способами: упрощение алгоритма, сохраняя при этом семантику, упрощение модели памяти, делая ее более пригодной для символьного исполнения и т. д.

USVM [6] — универсальная символьная машина для анализа программ на нескольких языках. К числу таких языков относится и Java, для которой написаны аппроксимации ее стандартной библиотеки. Однако в данный момент времени эти аппроксимации могут содержать множество ошибок, а реализовать тестовую систему весьма проблематично, так как она должна отслеживать расхождения в семантике между аппроксимациями и изначальным кодом.

Данная работа посвящена тестированию аппроксимаций стандартной библиотеки Java в USVM.

1. Постановка задачи

Целью работы является тестирование аппроксимаций стандартной библиотеки Java в USVM. Для ее выполнения были поставлены следующие задачи:

1. Провести обзор символьного исполнения, реализующего его USVM и использования в нем аппроксимаций.
2. Реализовать тестирующую систему для аппроксимаций в USVM.
3. Добавить тесты для имеющихся аппроксимаций.
4. Провести тестирование на добавленных тестах.

2. Обзор

2.1. Символьное исполнение

Символьное исполнение — популярная техника анализа программ. Его главной идеей является особая обработка значений, возникающих во время его работы: все значения, вводимые пользователем или получаемые из внешних источников, обозначаются неопределенными — символьными. Символьными также являются значения, для вычисления которых необходимы другие символьные (например, при сложении, вычитании и т. д.). При этом в ходе своей работы исполнение учитывает любую неопределенность, связанную с символьными константами, разделяясь на несколько веток, каждая из которых соответствует условиям, которым должны бы были удовлетворять переменные при конкретном исполнении, чтобы попасть в эту ветку (например, при достижении инструкции `if (x < 5)`, где `x` символьное, исполнение разделится на 2 ветки, одна из которых накладывает на `x` условие быть меньше 5, а вторая — больше или равным 5).

В каждый момент работы символьного исполнения его состояние однозначно описывается тройкой $(stmt, \sigma, \pi)$, где

- `stmt` — текущая инструкция
- σ — символьное хранилище, содержащее конкретные значения одних переменных, и символьные других
- π — накопившиеся условия на переменные, их соответствие которым привело исполнение в текущее состояние

Когда исполнение ветки достигает конца, запускается так называемый SMT-решатель [1], определяющий, можно ли подобрать такие конкретные значения для символьных переменных, что условие, накопившееся в π , будет неверным.

Одним из факторов, препятствующих эффективной реализации символьного исполнения, является взрыв количества путей. Например,

С. Канер в [7] описывает небольшую программу, состоящую из одного цикла и нескольких `if` операторов и имеющую свыше 100 триллионов возможных путей исполнения. Самыми частыми причинами резкого увеличения количества путей являются циклы и вызовы методов, присутствующие почти в каждой программе.

Другой популярный камень преткновения символьного исполнения — взаимодействие с внешним окружением. Под внешним окружением подразумевается как системное окружение (например, операционная система, под которой должен запускаться тестируемый код), так и программное (другие программы или части данной, с которыми текущий код взаимодействует). Его нельзя не учитывать при тестировании, так как некоторые ошибки могут проявиться только при взаимодействии нескольких программных компонент.

Негативное влияние обеих этих проблем можно попробовать ослабить при помощи аппроксимаций.

2.2. USVM и аппроксимации

USVM — универсальная символьная машина, в данный момент поддерживающая Java, Go, Python. Имеются аппроксимации для части классов, присутствующих в стандартной библиотеке. Исполнение начинается с перевода кода в трехадресный вид. Далее происходит его замена аппроксимациями при помощи JasoDB [3] — написанной на Java библиотеки, позволяющей анализировать Java байткод и сохранять результаты в специальной базе данных. При этом заменяются только методы классов, содержащиеся в аппроксимациях, остальные остаются нетронутыми, что позволяет не перегружать аппроксимации дублирующей логикой из оригинальных классов. Только после этого начинается исполнение символьной машины.

3. Описание решения

Архитектура тестовой системы представлена на рис. 1.

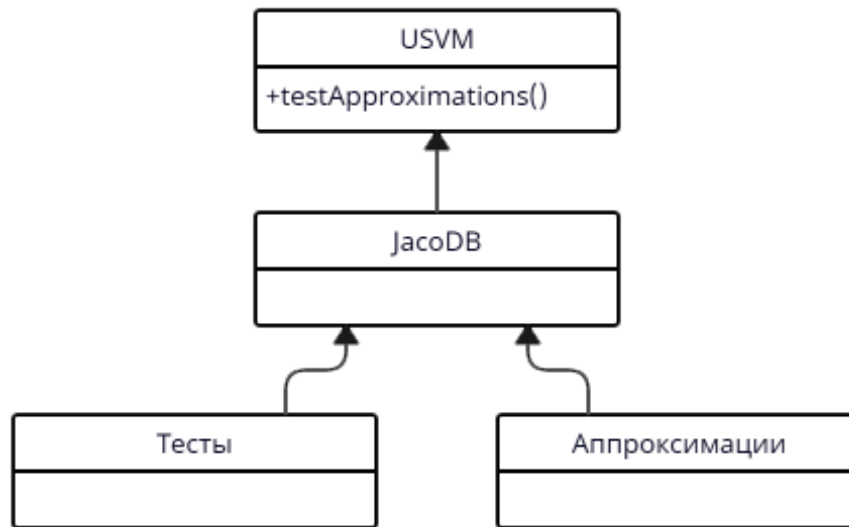


Рис. 1: Архитектура тестовой системы

USVM получает тесты и аппроксимации при помощи JacoDB, далее происходит замена тех классов, для которых присутствуют аппроксимации, и после этого начинается символьное исполнение на тестах.

3.1. Реализация тестовой системы

Каждый тест, а также классы, в которых содержатся тесты, помечаются специальной аннотацией `@Test`. Это позволяет отделять их от остальных методов и классов, поступающих через JacoDB. При этом каждый тест обязан удовлетворять общей структуре (см. листинг 1).

Листинг 1: Общая структура теста

```
@Test(executionMax = {значение})
public static int test(int execution) {
    // код теста
}
```

Значение `execution` рассматривается символьным. Это добавляет тесту немного «символьности», что позволяет удобно решать, например, следующие задачи:

- расширить область значений тех переменных, которые будут приравниваться `execution` (вместо 1 конкретного будет использоваться символьное)
- делать тест для всех коллекций из набора, что позволяет не дублировать код для каждой, и т. д.

Максимальное значение `execution` можно изменять при помощи `executionMax`, объявленного в интерфейсе аннотации. Таким образом, тест можно считать функцией от `execution` и полей класса, в котором находится метод теста, во множество $\{0, \dots, \text{executionMax}\} \vee \{-1\}$, при этом результат считается успешным, если он равен значению `execution`, с которым запускался тест.

3.2. Тесты

В качестве референса для тестов брались тесты из OpenJDK [4] и Guava [2]. Это было необходимо для того, чтобы семантика аппроксимированных классов полностью отражалась в тестах, которые, по сути, являются регрессионными. При этом тесты оптимизировались для символьного исполнения: приводились к требуемой структуре, разбивались для исполнения за приемлемое время символьным исполнением, заменялись на идентичные, не использующие неаппроксимированные классы и т. д.

4. Тестирование

Результаты тестирования приведены в таблице 1.

Таблица 1: Результаты тестирования

Коллекции	Кол-во тестов	Кол-во ошибок	Исправлено
AbstractCollection	2	1	0
Collection	7	0	0
AbstractList	8	4	4
ArrayList	5	2	1
LinkedList	4	2	1
List	76	4	4
AbstractMap	7	9	7
HashMap	11	3	3
LinkedHashMap	5	2	1
Optional	30	1	0
Random	12	0	0
Set	8	9	7
Splititerator	5	0	0

Суммарно было добавлено 180 тестов, найдено 37 ошибок в аппроксимациях, из которых было исправлено 28.

Заключение

В результате работы были выполнены следующие задачи:

1. Проведен обзор символьного исполнения, реализующего его USVM и использования в нем аппроксимаций.
2. Реализована тестирующая система для аппроксимаций в USVM.
3. Добавлены тесты для имеющихся аппроксимаций.
4. Проведено тестирование на добавленных тестах и исправлены некоторые найденные ошибки в аппроксимациях.

Репозиторий с кодом доступен на `GitHub`¹²³, имя пользователя: `IgorFilimonov`.

¹<https://github.com/UnitTestBot/java-stdlib-approximations/pull/15>

²<https://github.com/UnitTestBot/usvm/pull/217>

³<https://github.com/UnitTestBot/jacodb/pull/270>

Список литературы

- [1] Barrett Clark W., Kroening Daniel, Melham Thomas F. Problem Solving for the 21st Century: Efficient Solvers for Satisfiability Modulo Theories. — 2014. — URL: <https://api.semanticscholar.org/CorpusID:126248148>.
- [2] Guava. — URL: <https://github.com/google/guava>.
- [3] JacoDB. — URL: <https://github.com/UnitTestBot/jacodb>.
- [4] OpenJDK. — URL: <https://github.com/openjdk/jdk>.
- [5] A Survey of Symbolic Execution Techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia et al. // ACM Comput. Surv. — 2018. — Vol. 51, no. 3.
- [6] USVM. — URL: <https://github.com/UnitTestBot/usvm>.
- [7] Сэм Канер и Джек Фолк и Енг Кек Нгуен. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений. — «ДиаСофт», 2001.